Design and Analysis of Algorithms Notes

Computer Science Notes

UNIT I

Dictionaries: Sets, Dictionaries, Hash Tables, Open Hashing, Closed Hashing (Rehashing Methods), Hashing Functions (Division Method, Multiplication Method, Universal Hashing), Analysis of Closed Hashing Result (Unsuccessful Search, Insertion, Successful Search, Deletion), Hash Table Restructuring, Skip Lists, Analysis of Skip Lists.

SETS

- A set is a collection of well defined elements. The members of a set are all different.
- A set ADT can be defined to comprise the following operations:

Union(A, B, C): This operation combines all elements of set A and set B and places them in set C.

Ex: A={1,2,3,4} B={4,5,6} then C={1,2,3,4,5,6}

Intersection(A, B, C): This operation selects common elements from set A and set B (i.e., which appear both in A and B) and places them in set C.

Ex: A={1,2,3,4} B={4,5,6} then C={4}

Difference(A, B, C): This operation selects only the elements that appear in set A but not in set B and places them in set C.

Ex: A={1,2,3,4} B={4,5,6} then C={1,2,3}

Merge(A, B, C): This operation combines all elements of set A and set B and places them in set C and discards set A, B.

Ex: C={1,2,3,4,5,6}

Find(x): Returns the set name in which element 'x' is available.

Ex: A ={1,2,3}, B={4,5,6}
Find(5) returns set B

Member (x, A) or Search (x, A): Checks whether element' x' is available in 'A' or Not. If element is available returns search successful otherwise returns search unsuccessful.

Makenull (A): Makes a Set A to empty

Equal (A, B): Check whether two sets A and B are equal or not.

Ex: A={1,2,3} B={2,1,3} are equal

Assign (A, B): Assigns Set B elements to set A

Insert (x, A): Inserts an element 'x' into set A .Ex: Insert(7,A) now A={1,2,3,7}

Delete (x, A): Deletes an element 'x' from set A. Ex: Delete(3,A) now A={1,2,7}

Min (A): Returns minimum value from A . Ex: Min(A)=1.

- **A Set can be implemented with the following data structures**:
  (a) Bit Vector
  (b) Array
  (c) Linked List
    - Unsorted
    - Sorted

DICTIONARY

A *dictionary* is a container of elements from a totally ordered universe that supports the basic operations of inserting/deleting elements and searching for a given element. (OR)

Dictionary is a Dynamic-set data structure for storing items indexed using *keys*. It Supports operations Insert, Search, and Delete.

Ex: hash tables are dictionaries which provide an efficient implicit realization of a dictionary. Efficient explicit implementations include binary search trees and balanced search trees.

Dictionaries: A dictionary is a dynamic set ADT with the operations:

1. Makenull (D)
2. Insert (x, D)
3. Delete (x, D)
4. Search (x, D)

Dictionaries are useful in implementing symbol table of a compiler, text retrieval systems, database systems, page mapping tables, Large-scale distributed systems etc.

Dictionaries can be implemented with:

1. Fixed Length arrays
2. Linked lists: sorted, unsorted, skip-lists
3. Hash Tables: open, closed
4. Trees: Binary Search Trees (BSTs), Balanced BSTs like AVL Trees, Red-Black Trees Splay Trees, Multi way Search Trees like 2-3 Trees , B Trees,
5. Tries

- Let n be the number of elements in a dictionary D. The following is a summary of the performance of some basic implementation methods:

.	Search	Delete	Insert	Min
Array	O(n)	O(n)	O(n)	O(n)
Sorted linked list	O(n)	O(n)	O(n)	O(1)
Unsorted linked list	O(n)	O(n)	O(n)	O(n)

- Among these, **the sorted list** has the best **average case** performance
- Arrays, sorted linked lists, unsorted linked lists all takes O(n) time for insert, delete, Search, min operations

HASH TABLES

Necessity of Hash tables:

We can access any position of an array in constant time (i.e., in O(1)) . We think of the subscript as the key, and the value stored in the array as the data. Given the key, we can access the data in constant time.

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
10	20	25	30	35	40	45	50	55	60

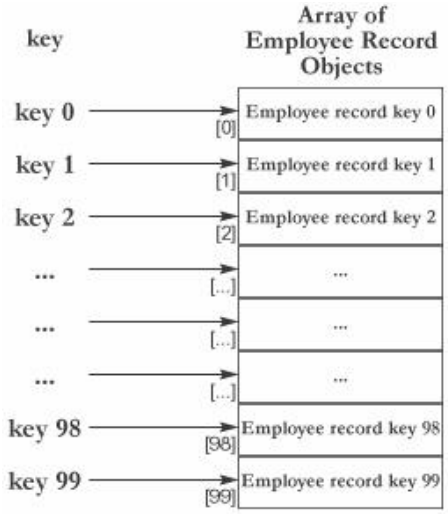
Generally search operation in an array takes O(n) time. But if we know the positions of each element where it was stored (i.e., at 1st location the element 20 was stored, at 6th location the element 45 was stored etc..) then we can directly access The element by moving to that position in O(1) time. This is the basic idea behind the implementation of hash tables.

Ex 1: For example, I used an array with size 10 to store the roll no.'s of 10 students. R.no. 1 was placed at 1st location, R.no. 2 was placed at 2nd location, R.no. 3 was placed at 3rd location and so on ..

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	2	3	4	5	6	7	8	9	10

Here I can access R.no. 9 directly from the 9th location, Rno:6 directly from 6th location. So for accessing the elements always it takes O(1) time. But in real cases this is not possible.

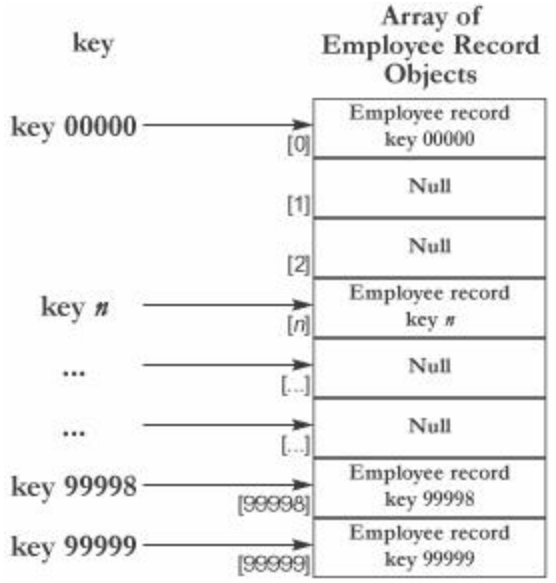
Ex 2: We have a list of employees of a fairly small company. Each of 100 employees has an ID number in the range 0 to 99. If we store the elements (employee records) in the array, then each employee's ID number will be an index to the array element where this employee's record will be stored. In this table an employee details can be accessed in O(1) time.



In this case once we know the ID number of the employee, we can directly access his record through the array index. There is a one-to-one correspondence between the element's key and the array index. In this an employee details can be accessed in O(1) time.

However, in practice, this perfect relationship is not easy to establish or maintain.

Ex 3: the same company might use employee's five-digit ID number as the key. With 5 digits the possible numbers are from 00000 to 99999. If we want to use the same technique as above, we need to set up an array of size 100,000, of which only 100 elements will be used (only 100 employees are working in that company).



In the above table only 100 locations are useful. Remaining 9900 records storage is wasted. This problem can be overcome with hash tables.

Ex 4: Sometimes companies maintain their employee numbers as alpha numeric letters. An employee ID may be SACET555 but we can't use this alphanumeric as an array index So we need something which maps a string or a large value into an array index.

HASH TABLES

(i) **Hash Table** is a data structure in which keys are mapped to array positions by a hash function. (Or) A Hash Table is a data structure for storing key/value pairs This table can be searched for an item in O(1) time using a hash function to form an address from the key.

(ii) Hash Function: Hash function is any well-defined procedure or mathematical function which converts a large, possibly variable-sized amount of data into a small datum, usually a single integer that may serve as an index into an array. (OR)

- Hash function is a function which maps key values to array indices. (OR)
- Hash Function is a function which, when applied to the key, produces an integer which can be used as an address in a hash table.
- We will use **h(k)** for representing the hashing function

(iii) Hash Values: The values returned by a hash function are called hash values or hash codes or hash sums or simply hashes

(iv) **Hashing** is the process of mapping large amount of data item to a smaller table with the help of a hashing function.

- Hash table is an extremely effective and practical way of implementing dictionaries.
- It takes O(1) time for search, insert, and delete operations in the **average case**. And O(n) time in the worst case.
- By careful design, we can make the probability that more than constant time is required to be arbitrarily small.
    - Hash Tables are two types they are: (a) Open or External (b) Closed or Internal

(c) Collisions: If x1 and x2 are two different keys, but the hash values of x1 and x2 are equal (i.e., h(x1) = h(x2)) then it is called a collision.

Ex: Assume a hash function = h(k) = k mod 10
h(19)=19 mod 10=9
h(39)=39 mod 10=9
here h(19)=h(39) This is called collision.

Collision resolution is the most important issue in hash table implementations. To resolve the collisions two techniques are there.

1. Open Hashing 2. Closed Hashing

Perfect Hash Function is a function which, when applied to all the members of the set of items to be stored in a hash table, produces a unique set of integers within some suitable range. Such function produces no collisions.
Good Hash Function minimizes collisions by spreading the elements uniformly throughout the array.

There is no magic formula for the creation of the hash function. It can be any mathematical transformation that produces a relatively random and unique distribution of values within the address space of the storage. Although the development of a hash function is trial and error, here are some hints that may make process easier:

- Set the size of the storage space to a prime number. This will help generate a more uniform distribution of addresses.
- Use modulo arithmetic (%). Transform a key in such a way that you can perform X % TABLE_SIZE to generate the addresses
- To transform a numeric key, try something like adding the digits together or picking every other digit.
- To transform a string key, try to add up the ASCII codes of the characters in the string and then perform modulo division.

(1) Open Hashing (OR) Separate Chaining:

- In this case hash table is implemented as an array of linked lists.
- Every element of the table is a pointer to a list. The list (chain) will contain all the elements with the same index produced by the hash function.
- In this technique the array does not hold elements but it holds the addresses of lists that were attached to every slot.
- Each position in the array contains a Collection of values of unlimited size (we use a linked implementation of some sort, with dynamically allocated storage).

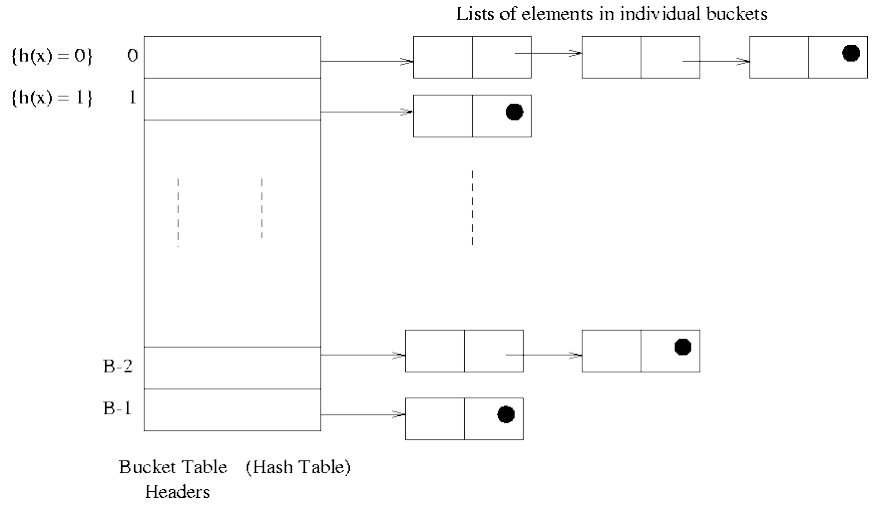
Here we will chain all collisions in lists attached to the appropriate slot. This allows an unlimited number of collisions to be handled and doesn't require a priori knowledge of how many elements are contained in the collection.
The tradeoff is the same as with linked lists versus array implementations of collections: linked list overhead in space and, to a lesser extent, in time.
Let U be the universe of keys. The Keys may be integers, Character strings, Complex bit patterns

- B the set of hash values (also called the buckets or bins). Let $B = \{0, 1, \ldots, m - 1\}$ where $m > 0$ is a positive integer.

A hash function h: U $\longrightarrow$ B associates buckets (hash values) to keys.

Ex: Collision Resolution by Chaining

Put all the elements that hash to the same value in a linked list. See Figure 1.1.



Example 1: See Figure 1.2. Consider the keys 0, 1, 2,4, 16, 25, 36, 49, 64, 81, 100.
Let the hash function be: **h(x) = x % 7**

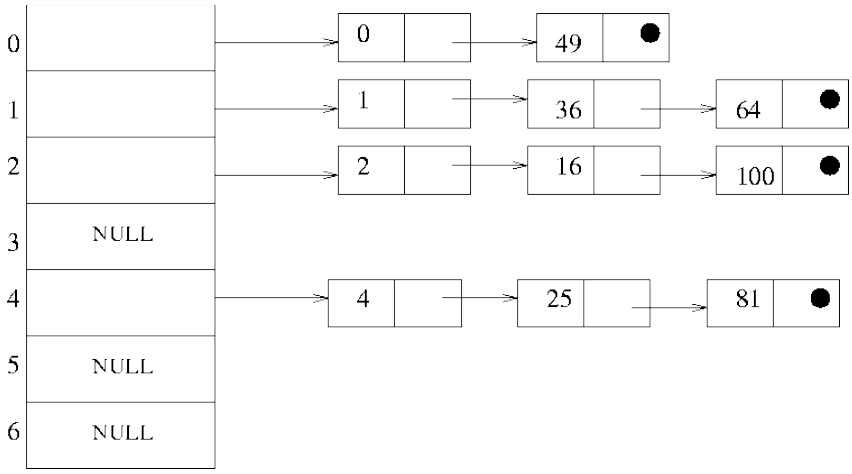


Figure 1.2: Open hashing: An example

Here list (chain) can be a sorted list or an unsorted list. If it is sorted list then the operations are easy.

Operations of Open Hashing or Separate Chaining:

- Search (x, T): Search for an element x in the list T[h(key (x))]
- Insert (x, T) : Insert x at the head of list T[h(key (x))]
- Delete (x, T) : Delete x from the list T[h(key (x))]

Searching

When it comes to search a value, the hash function is applied to the desired key K, and the array index is produced. Then the search function accesses the list to which this array element points to. It compares the desired key K, to the key in the first node. If it matches, then the element is found. In this case the search of the element is on the order of 1, O(1). If not, then the function goes through the list comparing the keys. Search continues until the element is found or the end of the list is detected. In this case the search time depends on the length of the list.

For example, In the above list if we want to search a key 100, then find the h(100). h(100)=100%7=2. So directly go to the 2nd location and search the first node has the element 100 if it is not 100 then go to the next location and check whether it is 100 do this until an element was found or last node is reached.

Algorithm for Separate chaining hashing: search

Open hashing is implemented with the following data structures

```
struct node
{
int k;
struct node *next;
};
struct node *r[10];
typedef struct node list;
```

```
list*search( key, r )
 {
list *p;
 p = r[ hashfunction(key) ];
 while ( p!=NULL && key!=p->k )
  p = p->next;
 return( p );
}
```

INSERTION: To insert a value with key K, find h(K) and add K to the collection(chain) in position h(K).

Algorithm for Separate chaining Hashing: Insertion

```
void insert( key, r )
   {
       int i;
     i = hashfunction( key );/*evaluates h(k)*/
     if (empty(r[i])) /*** insert in main array ***/
               r[i].k = key;
          else /*** insert in new node ***/
              r[i].next = NewNode( key, r[i].next );
     }
```

Deletion: To delete a value with key K, find h(k) and go to the array position h(k) search the list for the element K if it was found then remove it from the collection(chain) in position h(K). otherwise display that "Element was not displayed"

Advantages

1) The hash table size is unlimited. In this case you don't need to expand the table and recreate a hash function.
2) Collision handling is simple: just insert colliding records into a list.

Disadvantages

1) As the list of collided elements (collision chains) become long, search them for a desired element begin to take longer and longer.
2) Using pointers slows the algorithm: time required is to allocate new nodes.

Analysis of Open Hashing:

(i) Search:

(a) Best case: To search a Key K, first find out h(k) . Assume time to compute h(k) is O(1) and the Key 'K' is available as the first node then the complexity is O(1).

(b) Average case: Given hash table T with m slots holding n elements,

Load factor $\alpha = n/m$ = average keys per slot.

m – number of slots, n – number of elements stored in the hash table.

(i) Unsuccessful Search: *Uniform* hashing yields an average list length $\alpha = n/m$

- Any key not already in the table is equally likely to hash to any of the **m** slots.
- To search unsuccessfully for any key **k**, need to search to the end of the list T[h(k)], whose expected length is α.
- Adding the time to compute the hash function(1), the total time required is $O(1+\alpha)$.
- Search time for unsuccessful search is $O(1+\alpha)$

(ii) Successful Search:

- The probability that a list is searched is proportional to the number of elements it contains.
- Assume that the element being searched for is equally likely to be any of the **n** elements in the table.
- The number of elements examined during a successful search for an element **x** is 1 more than the number of elements that appear before x in x's list.
  - » These are the elements inserted after **x** was inserted.
  - » Goal: Find the average, over the n elements x in the table, of how many elements were inserted into x's list after x was inserted.
- Let x_i be the i^{th} element inserted into the table, and let $k_i = key[x_i]$.
- Define indicator random variables $X_{ij} = I\{h(k_i) = h(k_j)\}$, for all i, j.
- Simple uniform hashing $\Rightarrow \Pr\{h(k_i) = h(k_j)\} = 1/m$

$$\Rightarrow E[X_{ij}] = 1/m.$$

- Expected number of elements examined in a successful search is:

$$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right]$$

$$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right]$$

$$=\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}E[X_{ij}]\right)$$

$$=\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right)$$

$$=1+\frac{1}{nm}\sum_{i=1}^{n}(n-i)$$

$$=1+\frac{1}{nm}\left(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\right)$$

$$=1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right)$$

$$=1+\frac{n-1}{}$$

Expected total time for a successful search = Time to compute hash function + Time to search

$= O(2+\alpha/2 - \alpha/2n) = O(1+\alpha)$.

If n = O(m), then α=n/m = O(m)/m = O(1).

A successful search takes expected time O(1+α).
Searching takes constant time on average.

- Insertion is O(1) in the worst case.
- Deletion takes O(1) worst-case time when lists are doubly linked.
- Hence, all dictionary operations take O(1) time on average with hash tables with chaining.
- In the average case, the running time is $O(1 + \alpha)$,

It is assumed that the hash value h(k) can be computed in O(1) time. If n is O(m), the average case complexity of these operations becomes O(1) !

CLOSED HASHING (OR) OPEN ADDRESSING:

Rehashing

- Rehashing is resolving a collision by computing a new hash location (index) in the array.
- Re-hashing schemes use a second hashing operation when there is a collision. If there is a further collision, we re-hash until an empty "slot" in the table is found.
- The re-hashing function can either be a new function or a re-application of the original one. As long as the functions are applied to a key in the same order, then a sought key can always be located.

Closed Hashing (or) Open Addressing:

- In open addressing all elements stored in hash table itself. Each slot of a hash table contains either a key or NIL.
- It shows the way, when collisions occur, use a systematic (consistent) procedure to store elements in free slots of the table.
- Open addressing is the standard hash table implementation
- With *open addressing*, we store only one element per location, and handle collisions by storing the extra elements in other unused locations in the array.
- To find these other locations, we fix some *probe sequence* that tells us where to look if A[h(x)] contains an element that is not **x**.
- It is based on of resolving collisions by probing for free slots
- The hash formula determines the length and complexity of these probes.

Open addressing provides 3 different probing techniques.

1. Linear Probing
2. Quadratic Probing
3. Double Hashing

1. **Linear Probing:**
   - In case of linear probing, we are looking for an empty spot by incrementing the offset by 1 every time.
   - We explore a sequence of location until an empty one is found as follows:

> **h(x, i) = (h(x) + i) mod m**
>
> where **m** is the hash table size
> **and**
> **i** = 0, 1, 2, . . . , m-1

It is just h(x), h(x)+1, h(x)+2, ..., wrapping around at the end of the array. The idea is that if we can't put an element in a particular place, we just keep walking up through the array until we find an empty slot.

(a) Insertion(x,T): This operation inserts an element x into a hash table T, while inserting an element **if there is a collision** it applies linear Probing.
Procedure:

- First it evaluates h(x), if h(x) location is empty, then it places x into h(x).
- If there is a collision then it applies Linear Probing and locates another slot and if this slot is empty then it places the 'x' into that slot, Other wise it probes to another location, this procedure is repeated until an empty slot is found .
  - If there is no empty location in hash table then insertion is not possible

Example:

0	**1**	**2**	**3**	**4**
Empty	Empty	Empty	Empty	Empty

In the above hash table with m=5 insert the following values. 54, 20,44,33,21

Insert 54:h(54)=54 mod 5 = **4** *(so place 54 at 4^{th} location)*

0	**1**	**2**	**3**	**4**
Empty	Empty	Empty	Empty	54

Insert 20: h(20)=20 mod 5 = **0** *(so place 20 at 0^{th} location)*

0	**1**	**2**	**3**	**4**
20	Empty	Empty	Empty	54

Insert 44:
h(44)=44 mod 5 = **4** .(*but at 4^{th} location already an element is existed. So apply linear probing)*
h(44,1)=(h(44)+1)mod5 = (4+1)mod5 = **0** (*here also collision so probe to next location.)*
h(44,2)=(h(44)+2)mod5=(4+2)mod5=**1** *(1^{st} location is empty. So place 44 at 1^{st} location)*

0	**1**	**2**	**3**	**4**
20	44	Empty	Empty	54

Insert 33: h(33)=33 mod 5= **3** *(so place 33 at 3rd location)*

0	1	2	3	4
20	44	Empty	33	54

Insert 21:

h(21)=21 mod 5=**1** *(But at 1st location already an element is there*)

h(21,1)=(h(21)+1)mod5=(1+1)mod5= **2** *(2nd location is empty so place 21 at 2nd position)*

0	1	2	3	4
20	44	21	33	54

Algorithm for Linear Probing Hashing: Insertion

```
void insert( key, r[])
     { int n;
       int i, last;
     i = hashfunction( key ) ;/*computes h(x)*/
     last = (i+m-1) % m;
     while ( i!=last && !empty(r[i]) && !deleted(r[i]) && r[i]!=key )
           i = (i+1) % m;
     if (empty(r[i]) || deleted(r[i]))
      r[i] = key; /*** insert here ***/
     else Error     /*** table full, or key already in table ***/;
     }
```

(b) Search(x,T): Search operation searches for an element x in hash table 'T' and returns "search was successful" if the element was found. Other wise it returns "search was unsuccessful".

Procedure:

- First evaluates h(x) and examine slot h(x). Examining a slot is known as a probe.
- If slot h(x) contains key x, the search is successful. If the slot contains NIL, the search is unsuccessful.
- There's a third possibility: slot h(x) contains a key that is not x.
  - Compute the index of some other slot, based on x and which probe we are on.(Apply linear probing)
  - Keep probing until we either find key k or we find a slot holding NIL.

Algorithm for Linear Probing Hashing: Search

```
int search( key, r[] )
{
 int i, last;
     i = hashfunction( key ); /*computes h(x)*/
     last = (i+n-1) % m;
   while ( i!=last && !empty(r[i]) && r[i]!=key )
           i = (i+1) % m;
     if (r[i]==key)  return( i );
          else         return( -1 );
}
```

(c) Delete(x,T): This operation deletes the key x from hash table 'T'

Procedure:

- First evaluates h(x) and examine slot h(x). Examining a slot is known as a probe.

- If slot h(x) contains key x, then delete the element and make that location empty. Otherwise apply linear probing to locate the element. After linear probing also if the element was not found then display that "Deletion is impossible because the element was not in the table".

Advantages:

- All the elements (or pointer to the elements) are placed in contiguous storage. This will speed up the sequential searches when collisions do occur.
- It Avoids pointers;

Disadvantages:

Linear probing suffers from primary clustering problem

- Clustering: Element tend to cluster around elements that produce collisions. As the array fills, there will be gaps of unused locations.
  *Suffers from **primary clustering***:
  - Long runs of occupied sequences build up.
  - Long runs tend to get longer, since an empty slot preceded by *i* full slots gets filled next with probability (*i*+1)/*m*.
  - Hence, average search and insertion times increase.
- As the number of collisions increases, the distance from the array index computed by the hash function and the actual location of the element increases, increasing search time.
- The hash table has a fixed size. At some point all the elements in the array will be filled. The only alternative at that point is to expand the table, which also means modify the hash function to accommodate the increased address space.

2. **Quadratic Probing**: is a different way of rehashing. In the case of quadratic probing we are still looking for an empty location. However, instead of incrementing offset by 1 every time, as in linear probing, we will increment the offset by 1, 4,9, 16, ... We explore a sequence of location until an empty one is found as follows:

$\mathbf{h(x, i) = (h(x) + i^2) \bmod m}$
where **m** is the hash table size
and **i** = 0, 1, 2, . . . , m-1

(a) Insertion(x,T): This operation inserts an element x into a hash table T, while inserting an element if there is a collision it applies Quadratic Probing.

Procedure:

- First it evaluates h(x) if h(x) location is empty, then it places x into h(x).
- If there is a collision then it applies Quadratic Probing and locates another slot and if This slot is empty then it places the 'x' into that slot , Other wise it probes to another location, this procedure is repeated until an empty slot is found .
- If there is no empty location in hash table then insertion is not possible.

```
void insert( key, r[])
     { int n;
       int i, last;
     i = hashfunction( key ) ;/*computes h(x)*/
     last = (i+m-1) % m;
     while ( i!=last && !empty(r[i]) && !deleted(r[i]) && r[i]!=key )
           i = (i*i+1) % m;
     if (empty(r[i]) || deleted(r[i]))
      r[i] = key; /*** insert here ***/
     else Error     /*** table full, or key already in table ***/;
     }
```

For example consider a hash table with size 10. And insert the elements 59,18,49,58,28,21,33 into the list. (**E-indicates empty**).

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
E	E	E	E	E	E	E	E	E	E

Insert 59: h(59)=59mod 10=**9** *(place 59 at 9^{th} location*

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
E	E	E	E	E	E	E	E	E	59

Insert 18: h(18)=18mod 10=**8** *(place 18 at 8^{th} location).*

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
E	E	E	E	E	E	E	E	18	59

Insert 49: h(49)=49mod 10=**9** *(place 49 at 9^{th} location. But there is a collision so apply Q.P).*
*$h(49,T)=(h(49)+1^2)mod10=(9+1)mod10=$**0** (Place 49 at 0^{th} location)*

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
49	E	E	E	E	E	E	E	18	59

Insert 58: h(58)=58mod 10=**8** *(place 58 at 8^{th} location. But there is a collision so apply Q.P). $h(58,T)=(h(58)+1^2)mod10=(8+1)mod10=$**9** (collision at 9^{th} loc. So apply Q.P)*
*$h(58,T)= (h(58)+2^2)mod10=(8+4)mod10=$**2** (Place 58 at 2^{nd} location).*

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
49	E	58	E	E	E	E	E	18	59

Insert 28: h(28)=28mod 10=**8** *(place 28 at 8^{th} location. But there is a collision so apply Q.P).*
*$h(28,T)= (h(28)+1^2)mod10=(8+1)mod10=$**9** (collision at 9^{th} loc. So apply Q.P)*
*$h(28,T)= (h(28)+2^2)mod10=(8+4)mod10=$**2** (collision at 2^{nd} loc. So apply Q.P).*
*$h(28,T)= (h(28)+3^2)mod10=(8+9)mod10=$**7** (Place 28 at 7^{th} location).*

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
49	E	58	E	E	E	E	28	18	59

TutorialsDuniya.com

Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

Please Share these Notes with your Friends as well

Insert 21: h(21)=21mod 10=**1**(place 21 at 1st location)

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
49	21	58	E	E	E	E	28	18	59

Insert 33: h(33)=33mod 10=**3**(place 33 at 3rd location)

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
49	21	58	33	E	E	E	28	18	59

(b) Search(x,T): Search operation searches for an element x in hash table 'T' and returns "search was successful" if the element was found. Other wise it returns "search was unsuccessful".

Procedure:

- First evaluates h(x) and examine slot h(x). Examining a slot is known as a probe.
- If slot h(x) contains key x, the search is successful. If the slot contains NIL, the search is unsuccessful.
- There's a third possibility: slot h(x) contains a key that is not x.
  - Compute the index of some other slot, based on x and which probe we are on.(Apply Quadratic probing)
  - Keep probing until we either find key k or we find a slot holding NIL.

```
int search( key, r[] )
{
int i, last;
     i = hashfunction( key ); /*computes h(x)*/
     last = (i+n-1) % m;
   while ( i!=last && !empty(r[i]) && r[i]!=key )
           i = (i*i+1) % m;
     if (r[i]==key)  return( i );
      else     return( -1 );
 }
```

(c) Delete(x,T): This operation deletes the key x from hash table 'T'

Procedure:

- First evaluates h(x) and examine slot h(x). Examining a slot is known as a probe.
- If slot h(x) contains key x, then delete the element and make that location empty. Otherwise apply Quadratic probing to locate the element. After Quadratic probing also if the element was not found then display that "Deletion is impossible because the element was not in the table".

◆ **Disadvantage**: Can suffer from secondary clustering:

If two keys have the same initial probe position, then their probe sequences are the same.

3. Double Hashing: It uses two different hash functions

$h_1(x,i) = (h_1(x) + i\, h_2(x)) \bmod m$
where $h_1(x)$ is the first hash function and $h_2(x)$ is the second hash function, m is the hash table size, i=1,2,3,4 etc..

- Here h_1 and h_2 are two auxiliary hash functions**.** h_1 gives the initial probe. h_2 gives the remaining probes.
- In This $h_2(x)$ must be a relatively prime to m, so that the probe sequence is a full permutation of ⟨0, 1,…, m–1⟩.
- Choose *m* to be a power of 2 and have $h_2(x)$ always return an odd number. Or, Let *m* be prime, and have $1 < h_2(x) < m$.
- Suppose **$h_1(x)$**=x **mod** m then h_2 can be selected as **$h_2(x)$**=(R-x **mod** R) where R is a prime number nearer to m

Insert(x,T): This function inserts the key value 'x' into hash table 'T'.

Procedure:

- First it evaluates $h_1(x)$ if $h_1(x)$ location is empty, then it places x into $h_1(x)$.
- If there is a collision then it applies Quadratic Probing and locates another slot and if This slot is empty then it places the 'x' into that, Other wise it probes to another location, this procedure is repeated until an empty slot is found .
- If there is no empty location in hash table then insertion is not possible.

For example consider a hash table with size 10. And insert the elements 59,18,49,58, 21,33 into the list.(E-indicates empty).

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
E	E	E	E	E	E	E	E	E	E

Insert 59: $h_1(59)$=59mod 10=**9** *(place 59 at 9^{th} location*

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
E	E	E	E	E	E	E	E	E	59

Insert 18: $h_1(18)$=18mod 10=**8** *(place 18 at 8^{th} location).*

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
E	E	E	E	E	E	E	E	18	59

Insert 49: $h_1(49)$=49mod 10=**9** *(place 49 at 9^{th} location. But element is there so apply D.H).*
$h_1(49,T)=(h_1(49)+1.h_2(49))mod10$
***Where $h_2(49)$**=R- (x mod R)=(7- 49mod 7)=7-0=7(select R as 7 which is a prime nearer to m)*
*$h_1(49,T)=(h_1(49)+1.7))mod10=(9+7)$ mod 10=**6**(Place 49 at 6^{th} position).*

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
E	E	E	E	E	E	49	E	18	59

Insert 58: $h_1(58)=58 \bmod 10=$**8** *(place 58 at 8^{th} location. But there is a collision so apply D.H). $h_1(58,T)=(h_1(58)+1.h_2(58))mod10$*

Where $h_2(58)= R- (x \bmod R)=(7- 58mod\ 7)=7-2=5$

*$h_1(58,T)=(h_1(58)+1.5))mod10=(8+5) \bmod 10=$**3**(Place 58 at 3^{rd} position.)*

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
E	E	E	58	E	E	49	E	18	59

Insert 21: h(21)=21mod 10=1(place 21 at 1^{st} location)

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
E	21	E	58	E	E	49	E	18	59

Algorithm for Double Hashing: Insertion

```
void insert( key, r )
     {
       int i, inc, last;
     i = hash1(key) ;
     inc = hash2(key);
     last = (i+(m-1)*inc) % m;
     while ( i!=last && !empty(r[i]) && !deleted(r[i]) && r[i]!=key )
          i = (i+inc) % m;
     if ( empty(r[i]) || deleted(r[i]) )
               {
               /*** insert here ***/
               r[i] = key;
               n++;
               }
     else Error     /*** table full, or key already in table ***/;
     }
```

//Where hash1 (key) is the first hash function,hash2(key) is the second hash function .

(b) Search(x,T): Search operation searches for an element x in hash table 'T' and returns "search was successful" if the element was found. Other wise it returns "search was unsuccessful".

Procedure:

- First evaluates $h_1(x)$ and examine slot $h_1(x)$. Examining a slot is known as a probe.
- If slot $h_1(x)$ contains key x, the search is successful. If the slot contains NIL, the search is unsuccessful.
- There's a third possibility: slot $h_1(x)$ contains a key that is not x.
- Compute the index of some other slot, based on x and which probe we are on. (Apply Double hashing)
- Keep probing until we either find key k or we find a slot holding NIL.

Algorithm for Double Hashing: Search

```
int search( key, r )
     { int i, inc, last;

     i = hash1( key ) ;
     inc = hash2( key );
     last = (i+(n-1)*inc) % m;
     while ( i!=last && !empty(r[i]) && r[i]!=key )
          i = (i+inc) % m;
     if (r[i]==key)  return( i );
          else         return( -1 );
     }
```

//Where hash1 (key) is the first hash function,hash2(key) is the second hash function

(c) Delete(x,T): This operation deletes the key x from hash table 'T'

Procedure:

- First compute $h_1(x)$ value and examine slot $h_1(x)$. Examining a slot is known as a probe.
- If slot $h_1(x)$ contains key x, then delete the element and make that location empty. Otherwise apply Double Hashing to locate the element. After probing also if the element was not found then display that "Deletion is impossible because the element was not in the table".

Advantages: Distributes keys more uniformly than linear probing

A Comparison of Rehashing Methods

Linear Probing	m distinct probe sequences	Primary clustering
Quadratic Probing	m distinct probe sequences	No primary clustering; but secondary clustering
Double Hashing	m^2 distinct probe sequences	No primary clustering No secondary clustering

HASHING FUNCTIONS

Choosing a good hashing function, **h(k)**, is essential for hash-table based searching. **h** should distribute the elements of our collection as uniformly as possible to the "slots" of the hash table. The key criterion is that there should be a minimum number of collisions.

If the probability that a key, **k**, occurs in our collection is **P(k)**, then if there are **m** slots in our hash table, a *uniform hashing function*, **h(k)**, would ensure:

$$\sum_{k|h(k)=0} P(k) = \sum_{k|h(k)=1} P(k) = \ldots = \sum_{k|h(k)=m-1} P(k) = \frac{1}{m}$$

Sometimes, this is easy to ensure. For example, if the keys are randomly distributed in (0,**r**], then,

h(k) = floor((mk)/r) will provide uniform hashing.

Mapping keys to natural numbers

Most hashing functions will first map the keys to some set of natural numbers, say (0,r]. There are many ways to do this, for example if the key is a string of ASCII characters, we can simply add the ASCII representations of the characters mod 255 to produce a number in (0,255) - or we could **xor** them, or we could add them in pairs mod 2^{16}-1, or ...

Having mapped the keys to a set of natural numbers, we then have a number of possibilities.

1. MOD FUNCTION:

h(k) = k mod m.

When using this method, we usually avoid certain values of **m**. Powers of 2 are usually avoided, for **k mod 2^b** simply selects the **b** low order bits of **k**. Unless we know that all the 2^b possible values of the lower order bits are equally likely, this will not be a good choice, because some bits of the key are not used in the hash function.

Prime numbers which are close to powers of 2 seem to be generally good choices for **m**.

For example, if we have 4000 elements, and we have chosen an overflow table organization, but wish to have the probability of collisions quite low, then we might choose **m** = 4093. (4093 is the largest prime less than $4096 = 2^{12}$.)

2. MULTIPLICATION METHOD:

- Multiply the key by a constant **A**, $0 < A < 1$,
- Extract the fractional part of the product,
- Multiply this value by **m**.

Thus the hash function is:

h(k) = floor(m * (kA - floor(kA)))

In this case, the value of **m** is not critical and we typically choose a power of 2 so that we can get the following efficient procedure on most digital computers:

- Choose $m = 2^p$.

- Multiply the **w** bits of **k** by **floor(A * 2^w)** to obtain a 2**w** bit product.
- Extract the **p** most significant bits of the lower half of this product.

It seems that: **A** = (sqrt(5)-1)/2 = 0.6180339887 is a good choice (*see* Knuth, "Sorting and Searching", v. 3 of "The Art of Computer Programming").

3. UNIVERSAL HASHING

This involves choosing a hash function randomly in a way that is independent of the keys that are actually going to be stored. We select the hash function at random from a carefully designed class of functions.

- Let Φ be a finite collection of hash functions that map a given universe U of keys into the range {0, 1, 2,..., m - 1}.
- Φ is called **universal** if for each pair of distinct keys $x, y \in U$, the number of hash functions $h \in \Phi$ for which $h(x) = h(y)$ is precisely equal to

$$\frac{|\Phi|}{m}$$

- With a function randomly chosen from Φ, the chance of a collision between x and y where $x \neq y$ is exactly **1/m**.

Example of a universal class of hash functions:

Let table size m be prime. Decompose a key x into r +1 bytes. (i.e., characters or fixed-width binary strings). Thus

$$x = (x_0, x_1,..., x_r)$$

Assume that the maximum value of a byte to be less than m.
Let $a = (a_0, a_1,..., a_r)$ denote a sequence of $r + 1$ elements chosen randomly from the set {0, 1,..., m - 1}. Define a hash function $h_a \in \Phi$ by

$$h_a(x) = \sum_{i=0}^{r} a_i x_i \bmod m$$

With this definition, $\Phi = \bigsqcup_a \{h_a\}$ can be shown to be universal. Note that it has m^{r+1} members.

SKIP LISTS

The worst case search time for a sorted linked list is O(n), as we can only traverse the list and cannot skip nodes while searching. For a balanced binary search tree, we skip almost half of the nodes after one comparison with root. For a sorted array, we have a random access and we apply binary search on arrays.

Can we augment sorted linked list to make the search faster? The answer is “skip list”.

The idea is simple; we create multiple layers so that we can skip some nodes. See the example with 16 nodes, and two layers. The upper layer works as “Express lane” which connects only to main outer stations and the lower layer works as an “Normal Lane” which connects every station. Suppose if we want to search for a key it is carried out as follows:

We start from the first node of the express lane, and keep moving on express lane till we find a node whose next is greater than key. Once we find such a node on the express lane, we move to normal lane and linearly search for key on the normal lane’

ANALYSIS: What is the complexity with two layers?

The worst case complexity is the number of nodes on "express lane" plus the number of nodes in a segment. Here a segment is number of nodes between two "express lane" nodes. So if we have "n" nodes on the normal lane, we have square root(n) nodes on express lane;

n=16 (nodes on normal lane), square root(n) = 4 (nodes on express lane)

We equally divide the normal lane, then there will be square root(n) nodes in every segment of "normal lane". Square root(n) is actually optimal division with two layers. With this arrangement, the number of nodes traversed for a search will be O(square root(n))

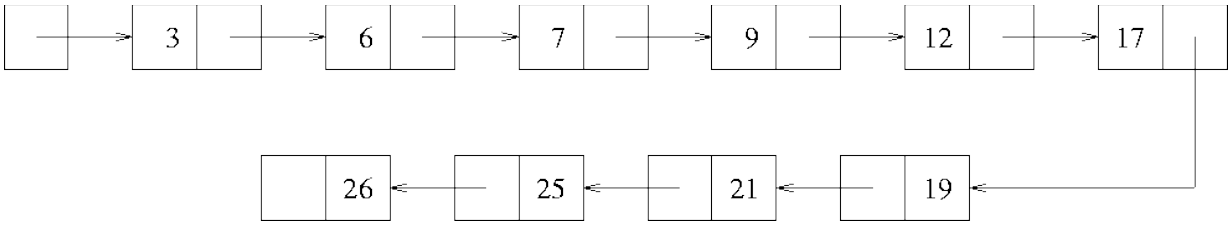
Can we do better?

The time complexity of skip list can be reduced further by adding more layers. In fact, time complexity of insert, delete, and search can be O(logn).

Skip List is a probabilistic alternative to balanced trees.

- Skip lists use probabilistic balancing rather than strictly enforced balancing.
- Although skip lists have bad worst-case performance, no input sequence consistently produces the worst-case performance (like quicksort).
- It is very unlikely that a skip list will be significantly unbalanced. For example, in a dictionary of more than 250 elements, the chance is that a search will take more than 3 times the expected time $\leq 10^{-6}$.
- Skip lists have balance properties similar to that of search trees built by random insertions, yet do not require insertions to be random.

Figure 1.4: A singly linked list



Consider a singly linked list as in Figure 1.4. We might need to examine every node of the list when searching a singly linked list.

Figure 1.5 is a sorted list where every other node has an additional pointer, to the node two ahead of it in the list. Here we have to examine no more than $\left\lceil \frac{n}{2} \right\rceil$ +1 nodes.

Figure 1.5: Every other node has an additional pointer

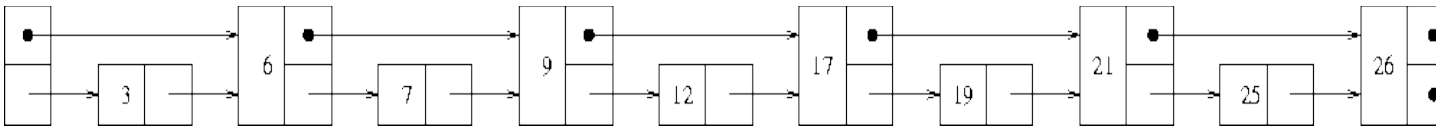
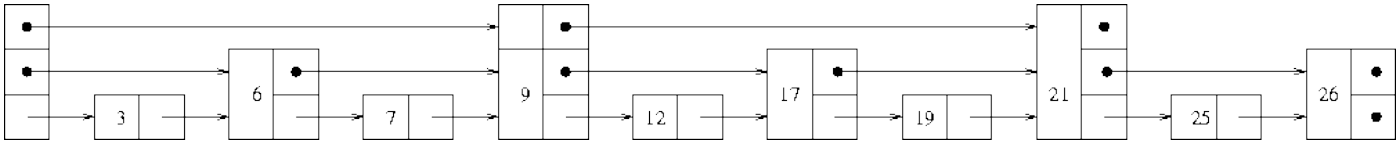


Figure 1.6: Every second node has a pointer two ahead of it



In the list of Figure 1.6, every second node has a pointer two ahead of it; every fourth node has a pointer four ahead if it. Here we need to examine no more than $\lceil \frac{n}{4} \rceil$ + 2 nodes.

In Figure 1.7, (every $(2^i)^{th}$ node has a pointer (2^i) node ahead (i = 1, 2,...); then the number of nodes to be examined can be reduced to $\lceil \log_2 n \rceil$ while only doubling the number of pointers.

Figure 1.7: Every $(2^i)^{th}$ node has a pointer to a node (2^i) nodes ahead (i = 1, 2,...)

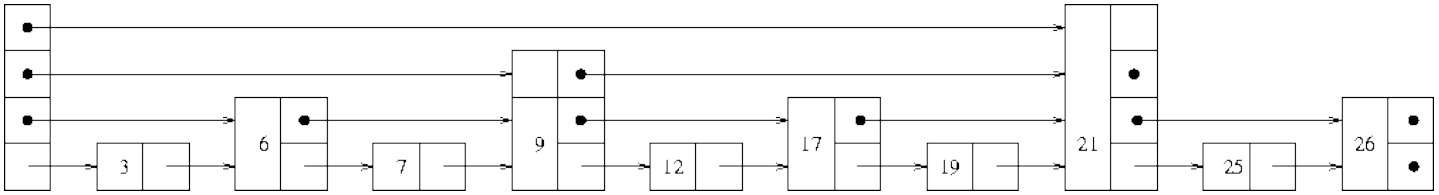
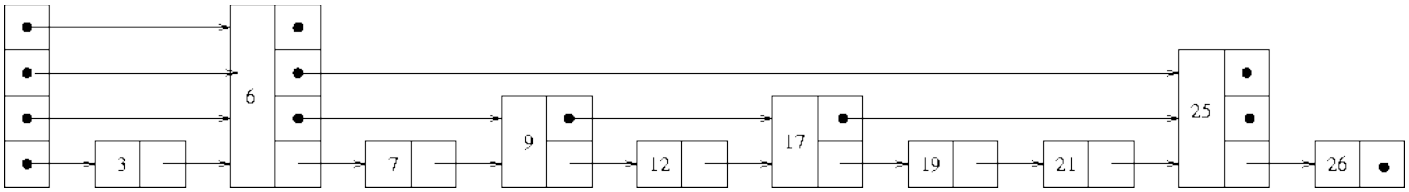


Figure 1.8: A skip list



- A node that has k forward pointers is called a *level k* node. If every $(2^i)^{th}$ node has a pointer (2^i) nodes ahead, then

  # of level 1 nodes 50 %
  # of level 2 nodes 25 %
  # of level 3 nodes 12.5 %

- Such a data structure can be used for fast searching but insertions and deletions will be extremely cumbersome, since levels of nodes will have to change.

- What would happen if the levels of nodes were randomly chosen but in the same proportions (Figure 1.8)?
  - level of a node is chosen randomly when the node is inserted
  - A node's i^{th} pointer, instead of pointing to a node that is 2^{i-1} nodes ahead, points to the next node of level *i* or higher.
  - In this case, insertions and deletions will not change the level of any node.
  - Some arrangements of levels would give poor execution times but it can be shown that such arrangements are rare.

  Such a linked representation is called a skip list.

- Each element is represented by a node the level of which is chosen randomly when the node is inserted, without regard for the number of elements in the data structure.
- A level *i* node has *i* forward pointers, indexed 1 through *i*. There is no need to store the level of a node in the node.
- Maxlevel is the maximum number of levels in a node.
  - Level of a list = Maxlevel
  - Level of empty list = 1
  - Level of header = Maxlevel

Initialization:

An element NIL is allocated and given a key greater than any legal key. All levels of all lists are terminated with NIL. A new list is initialized so that the level of list = maxlevel and all forward pointers of the list's header point to NIL.

Search:

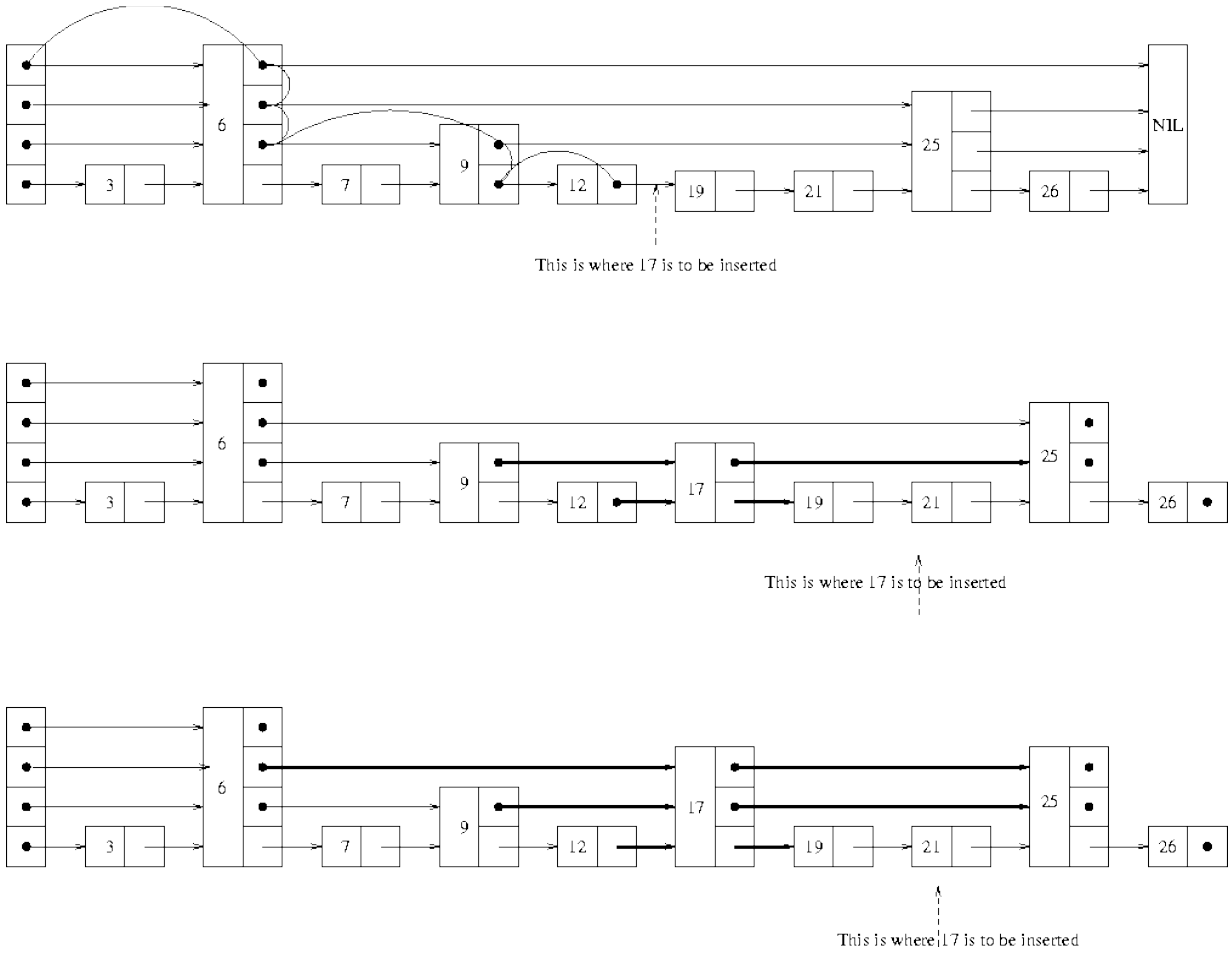
We search for an element by traversing forward pointers that do not overshoot the node containing the element being searched for. When no more progress can be made at the current level of forward pointers, the search moves down to the next level. When we can make no more progress at level 1, we must be immediately in front of the node that contains the desired element (if it is in the list).

Insertion and Deletion:

- Insertion and deletion are through search and splice
- update [*i*] contains a pointer to the rightmost node of level *i* or higher that is to the left of the location of insertion or deletion.
- If an insertion generates a node with a level greater than the previous maximum level, we update the maximum level and initialize appropriate portions of update list.
- After a deletion, we check to see if we have deleted the maximum level element of the list and if so, decrease the maximum level of the list.
- Figure 1.9 provides an example of Insert and Delete. The pseudo code for Insert and Delete is shown below.

```
search (list, searchkey)
      { x = list → header; for (i = list → level; i >= 1; i - -) {
        while (x → forward[i] → key < searchkey)
        x = x → forward[i];
   }
        x = x → forward[i];
      if (x → key = searchkey) return (true)
        else return false;
}
```

Figure 1.9: A skip list




```
insert (list, searchkey);
      { x = list → header ;
        for (i = list → level; i ≥ 1; i - -) {
          while (x → forward[i] → key < searchkey)
        x = x → forward[i];
        update[i] = x
      }
       x = x → forward[1];
       if (x → key = searchkey) return (``key already present")
          else {
       newLevel = randomLevel( );
          if newLevel > list → level {
            for (i = list → level + 1; i ≤ newLevel; i ++ )
          update[i] = list → header;
      }
       x = makenode(newLevel, searchkey);
          for (i = 1, i ≤ newLevel; i++) {
       x → forward[i] = update[i] → forward[i];
       update[i] → forward[i] = x
      }
}}
```

```
delete (list, searchkey);
    { x = list → header ;
      for (i = list → level; i ≥ 1; i - ) {
        while (x → forward[i] → key <searchkey)
      x = x → forward[i];
        update[i] = x
    }
      x = x → forward[1];
      if (x → key = searchkey) {
        for (i = 1; i ≤ list → level; i ++) {
      if (update[i] → forward[i] ≠ x) break;
      if (update[i] → forward[i] = x →forward[i];
}
    free(x)
      while (( list → 1) && (list → header → forward [list+level] = NIL))
            list → level = list → level – 1
```

Analysis of Skip Lists

In a skip list of 16 elements, we may have

- 9 elements at level 1
- 3 elements at level 2
- 3 elements at level 3
- 1 element at level 6

- One important question is:
  Where do we start our search? Analysis shows we should start from level $L(n)$ where
  $$L(n) = \log_2 n$$
  In general if p is the probability fraction,
  $$L(n) = \log_{\frac{1}{p}} n$$
  where p is the fraction of the nodes with level i pointers which also have level $(i + 1)$ pointers.

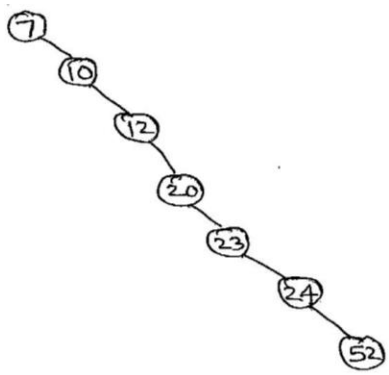
- However, starting at the highest level does not alter the efficiency in a significant way.
- Another important question to ask is:
  What should be MaxLevel? A good choice is
  $$MaxLevel = L(N) = \log_{\frac{1}{p}} N$$
  where N is an upper bound on the number of elements is a skip list.

- Complexity of search, delete, insert is dominated by the time required to search for the appropriate element. This in turn is proportional to the length of the search path. This is determined by the pattern in which elements with different levels appear as we traverse the list.
- Insert and delete involve additional cost proportional to the level of the node being inserted or deleted.

UNIT-2

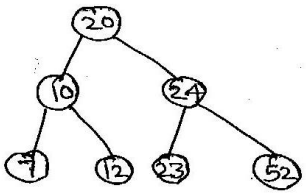
BALANCED TREES

Consider the following binary search tree



In this BST
Locating 7 takes 1 comparison
Locating 10 takes 2 comparisons
Locating 3 takes 3 comparisons
Locating 52 takes 7 comparisons
In this tree to locate nth element 'n' comparisons are required.

- So the search effect for this particular BST is O(n).
- These types of trees are said to be as unbalanced trees.
- With the same elements consider another tree



In this BST

- Locating 20 requires 1 comparison
- Locating 10,24 requires 2 comparisons
- Locating 7,12,23,52 requires 3 comparisons

- The maximum search effort of this tree is 3 i.e., the search effort is O(log n). These types of trees are called balanced trees.
- In the above mentioned trees, when compared to unbalanced trees, in a balanced tree the number of comparisons are reduced from '7' to '3'.
- For a completely unbalanced tree with 1000 nodes, the worst case comparisons are 1000, whereas the worst case comparisons for a balanced tree with 1000 nodes will be around 10.

DEFINITION (1): An empty tree is height balanced.

If 'T' is a non-empty binary tree with T_L and T_R as its left and right sub trees respectively, then 'T' is height balanced if and only if

(1) T_L and T_R are height balanced and

(2) $|h_L-h_R|\leq 1$, where h_L and h_R are the heights of left sub tree (T_L) and right sub tree (T_R) respectively.

DEFINITION (2): Balanced trees are trees whose height in the worst case is O(logn) and for which the operations INSERT, DELETE and SEARCH can be implemented in O(logn) time.

- There exist over one hundred types of balanced search trees. Some of the more common types are: AVL trees, 2-3 trees, red-black trees, Splay trees, B-trees etc.

HEIGHT: For all trees the heights are considered from leaf nodes. For the leaf nodes height is 1 and for their parents the height is 2 and so on.

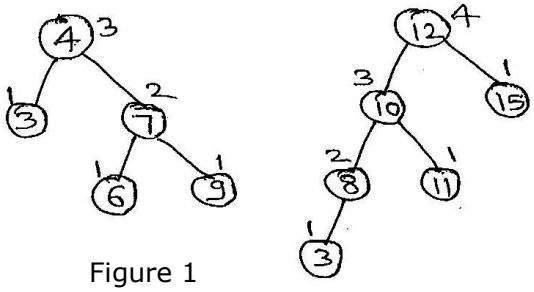


Figure 1

In figure 1, h(4)=3, h(3)=1, h(7)=2, h(9)=1.

6, 9, 3 are leaf nodes so height is 1.

AVL TREE DEFINITION: An AVL tree is a binary search tree in which:

1. The heights of the right sub tree and left sub tree of the root differ by at most 1.
2. The left sub tree and right sub tree are themselves AVL trees.

Maximum depth of AVL tree with 'n' nodes is O(logn).

AVL TREES

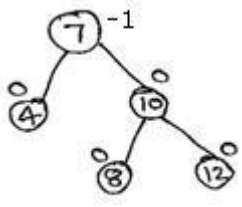
- In 1962 Adelson-Velskii and Landis introduced a balanced binary tree with respect to the heights of sub trees.
- An AVL tree is a height balanced binary search tree.
- Because of its balanced nature insertion, deletion, search operation for an AVL tree of 'n' nodes takes only O(logn) time.

AVL TREE BALANCE FACTOR

- The balance factor of a node 'T' in a binary tree is defined to be h_L-h_R, where h_L and h_R are heights of left and right sub trees of 'T', i.e., BF(T) = h_L-h_R.
- For any node in an AVL tree, the balance factor should be either -1 or 0 or 1.

Examples of AVL Trees

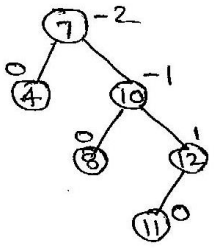
- In this tree for node 4, there is no left & right sub trees, so BF(4)=0-0=0
- For node 7 only left sub tree is there, so BF(7)=h_L-h_R=1-0 =1 (height of node 4 is 1)
- The BF(4), BF(7) both are in range -1,0,1. So this tree is an AVL tree.



In this example: h(8)=1, h(12)=1, h(10)=2, h(7)=3, h(4)=1.
BF(8)=0-0=0 (Because no left & right sub trees)
BF(12)=0-0=0 (Because no left & right sub trees)
BF(10)=h(8)-h(12)=1-1=0
BF(4)=0-0=0 (Because no left & right sub trees)
BF(7)=h(4)-h(10)=1-2=-1

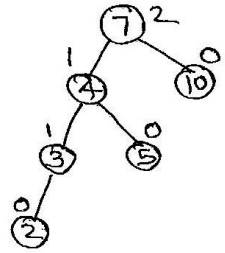
All balance factors are in the range -1,0,1. So this is also an AVL tree.

NON-AVL TREES



This tree is not an AVL tree.
h(11)=1, h(12)=2, h(10)=3, h(7)=4, h(8)=1, h(4)=1.
BF(11)=0-0=0 (no left & right sub trees)
BF(12)=1-0=1 (only left sub tree with height 1)
BF(8)=0-0=0 (no left & right sub trees)
BF(10)=h(8)-h(12)=1-2=-1
BF(4)=0-0=0 (no left & right sub trees)
BF(7)=h(4)-h(10)=1-3=-2

The root node has a balance factor of -2, but in AVL tree every node should have BF of 0, -1 or 1. So this is not an AVL tree.



This is not an AVL tree because the root has a BF of 2.

NOTE

- Balance Factor -1 indicates the left sub tree height is less than right sub tree.
- Balance Factor 1 indicates right sub tree height is less than left sub tree.
- Balance Factor 0 indicates that left and right sub trees height are equal.

(or)

BF(n)= + indicates left sub tree is heavier.
BF(n)= - indicates right sub tree is heavier.
BF(n)= 0 indicates balanced.

OPERATIONS ON AVL TREE

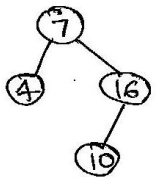
We can apply 3 operations on AVL tree. They are:

(a) Insert(X,T): Insert an element 'X' into 'T'.
(b) Delete(X,T): Delete an element 'X' from 'T'.
(c) Search(X,T): Search and locate the element 'X' in 'T'.

INSERTIONS AND DELETIONS ON AVL TREES

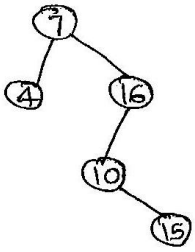
Insertion: Inserting an element into an AVL tree is similar to inserting an element into binary search tree.

- Insert a node at leaf level and if the element to be inserted is less than root node, insert this element in the left sub tree otherwise insert the element in the right sub tree.
- Apply this rule recursively for all nodes in the tree.



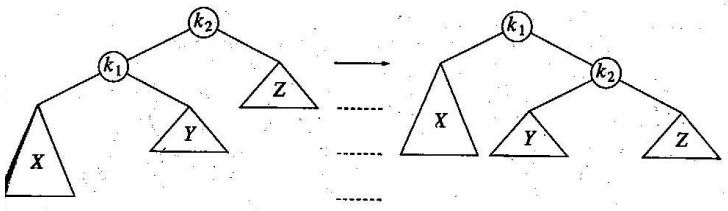
In this tree we want to insert '15'. 15 is greater than 7 so it will be inserted in the right sub tree of root and '15' is less than '16' so it is inserted in left sub tree of 16, but it is greater than 10, so 15 is inserted as the right child of 10.

- In AVL tree & Binary Search Tree the insertion procedure is same.



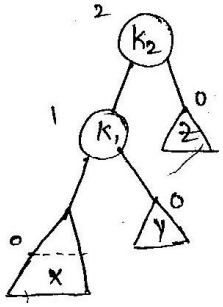
- When an insertion operation is applied to an AVL tree, we need to update all the balancing information for the nodes on the path back to the root.
- As we follow the path up to the root and update the balancing information, we may find a node whose new balance may violate the AVL condition.
- When the AVL condition is violated, the tree is no more an AVL tree. So we have to rebalance the tree such that AVL condition is preserved.
- The technique that is used to rebalance the AVL tree is either performing 1) Single rotation or (2) Double rotation.
- Let us call the node that must be rebalanced as '∝'.
- Since any node has at most two children, and a height imbalance requires that ∝'s two sub trees height differ by two, it is easy to see that violation may occur in 4 cases:
    1. An insertion into the left sub tree of the left child of ′∝′.
    2. An insertion into the right sub tree of the right child of ′∝′.
    3. An insertion into the left sub tree of the right child of ′∝′.
    4. An insertion into the right sub tree of the left child of ′∝′.
- The cases 1, 2 in which the insertion occurs on the "outside" (i.e., left-left, right-right) is fixed by a single rotation of the tree.
- The cases 3, 4 in which the insertion occurs on the "inside" (i.e., left-right, right-left) is fixed by double rotation.

SINGLE ROTATION (LEFT – LEFT)



Before **After**

AVL trees (Single Rotation-LL)

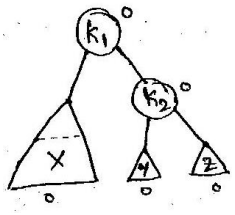


- In this diagram the property of an AVL tree is violated at node 'k2' (BF=2).
- This was because a node is inserted at the sub tree of 'X'
- The node was inserted to the left child of k2 (i.e., k1). For k1 again in the left sub tree. So this problem is called as 'LL' and it can be fixed by single rotation.
- Before rotation the relationships among all the values in the tree are

  k1<k2, X<k1, Y>k1, Z>k2

  X<k2, Y<k2

After Rotation



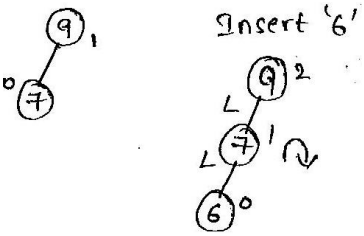
- After rotation also all the above relationships are preserved.
- ('Z' is in same position.

  'k2' becomes as right child of k1.

  'Y' becomes as left child of k2.

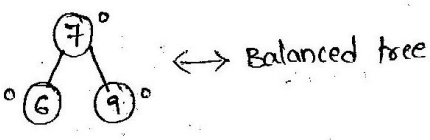
  'X' is in same position)

Example 1: 9,7,6



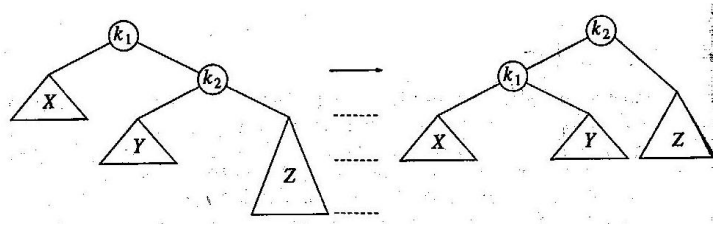
- Because of adding '6' AVL property is violated at '9'.
- 6 was inserted to the left child of '9' (i.e., 7) and for '7' also in left sub tree. So apply LL rotation.

Set k2->left=k1->right K1->right=k2.



K1->left, K2->right remains in same position.

SINGLE ROTATION (RIGHT - RIGHT)



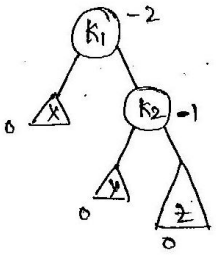
Before **After**

TutorialsDuniya.com

Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

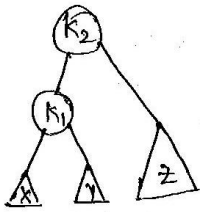
- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

Please Share these Notes with your Friends as well



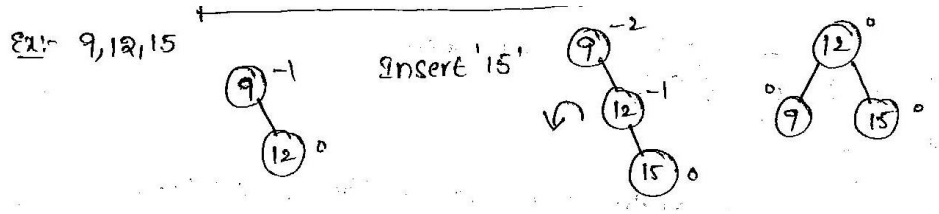
- In this tree, because of inserting a node in 'Z' sub tree the AVL property is violated at node k1.
- For node k1 the insertion is made to its right child 'k2'. For 'k2' again the insertion is made in its right sub tree.
- This problem is called as 'RR' and fixed by single rotation.

X<k1, k1<k2, Y<k2, Z>k1, Y>k1, Z>k2.

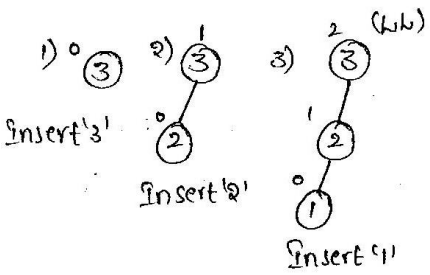


- After rotation, all the above relationships are preserved.
- k1's left child position and k2's right child position are not changed.
- k2's left child becomes as k1's right child.
- 'k1' becomes as k2's left child.

Set: k1->right=k2->left; K2->left=k1



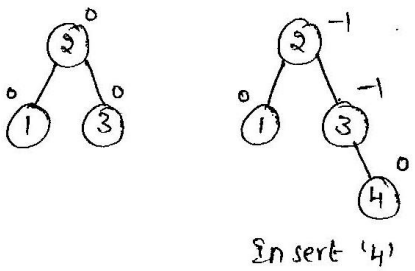
Insert the following elements into an AVL tree: 3,2,1,4,5,6,7

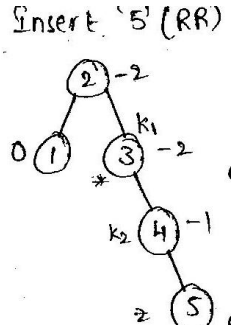


It is not an AVL tree. AVL property is violated at node '3' because of insertion of '1'

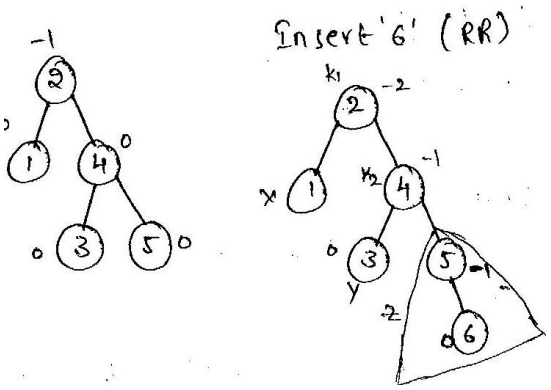
For '3', '1' is inserted in LL position.

So apply LL rotation.

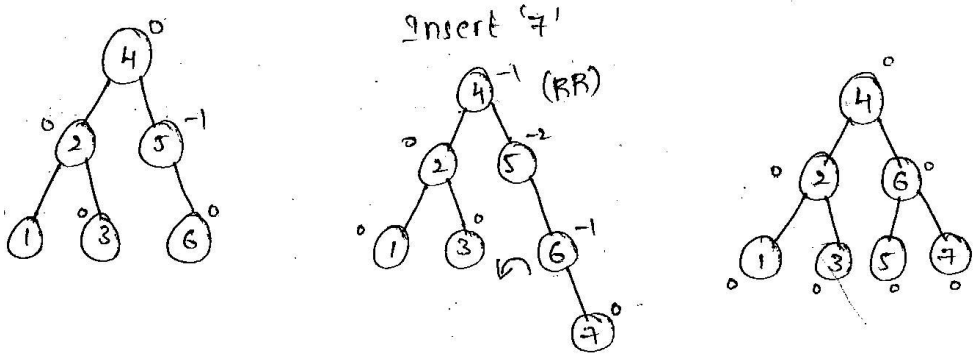




In this tree, 2 nodes are there with BF=-2. But consider the node from leaf, '3' (i.e., near to leaf level). Insertion was made in RR position.



At node '2' the AVL property was violated. For node '2' the insertion is made at RR position.



DOUBLE ROTATION

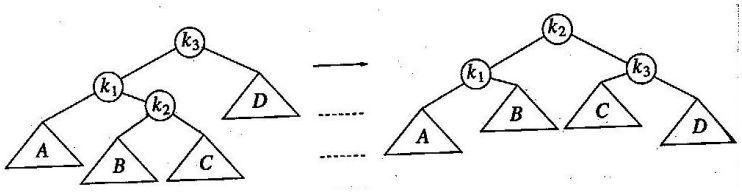
Assume '∝' is a node that is not satisfying the AVL property (i.e., the BF is not 0, 1, or -1) because of the following insertions:

1. An insertion into the right sub tree of the left child '∝'.

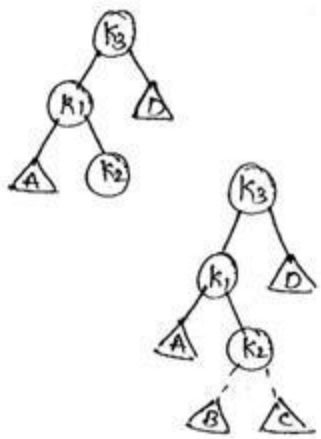
2. An insertion into the left sub tree of the right child of '∝'.

We can fix these two cases with double rotation.

LEFT-RIGHT (LR Rotation)



Before **After**

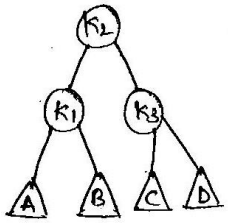


This tree is balanced. But if we insert the child for k2 (i.e., either 'B' or 'C') then the tree become unbalanced at node k3.

- Here the insertion was made to left child of k3 (i.e., k1) and for k1 the insertion was made in right sub tree. So it is called LR imbalance and fixed by double rotation.
- Before rotation the relationship among all elements in the tree are (all these should be preserved after rotation also):

k1<k3, k1<k2	k2>k1, k2<k3	k3>k1, k3>k2
A<k1	B<k2, B>k1	C>k2, C<k3

After rotation also the relations must be preserved.

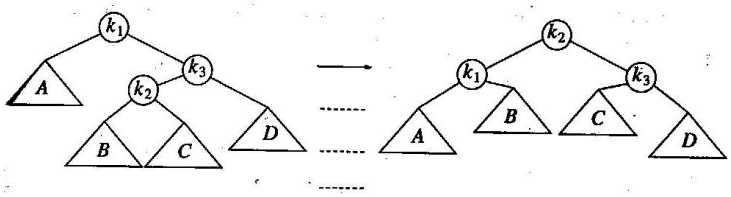


- The positions of 'A' and 'D' are not changed
- 'B' becomes as left child of k1 (B>k1)
- 'C' becomes as left child of k3 (C<k3)
- K2 becomes as parent.
- K1 becomes as left child of k2. (k1<k2).
- K3 becomes as right child of k2. (k3>k2).

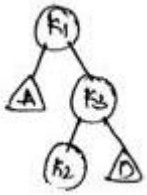
Set:

```
k1->right=k2->left;
k3->left=k2->right;
k2->left=k1;
k2->right k3.
```

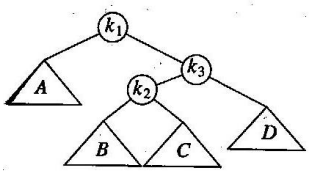
RIGHT-LEFT (RL Rotation)



Before **After**



This tree is balanced. But if we insert a child to k2 (i.e., B or C) then the AVL property is violated at node k1).

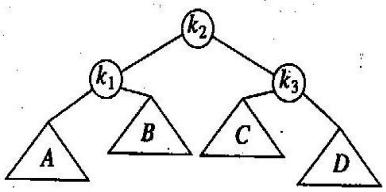


In this tree the insertion was made to right child of k1 (i.e., k3) and for k3 the insert was made in left sub tree. So it is called RL imbalance.

- Before rotation the relationships among all elements of the tree are:

k1<k3	k2<k3	k3>k1	B<k2	C>k2
k1<k2	k2>k1	k3>k2	B>k1	C>k1
			B<k3	C<k3

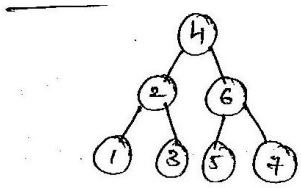
- k2's children will be adjusted as child for k1 and k3.
- After rotation also all these conditions should be preserved.



- k1's left child 'A', k3's right child 'D' will remain in the same position.
- B becomes as right child of k1 as B>k1.
- C becomes as left child of k3 as C<k3.
- k1 becomes as k2's left child.
- k3 becomes as k2's right child.

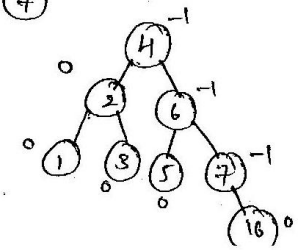
```
k3->left=k2->right;

k1->right=k2->left;

k2->left=k1;

k2->right= k3;
```

Example:

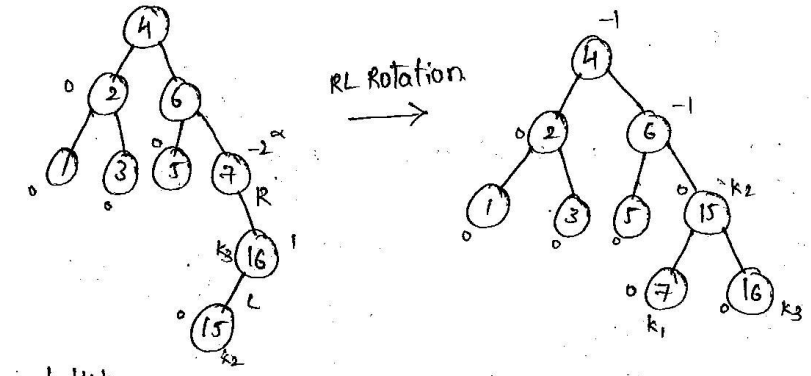


Insert the following list of elements into AVL Tree. 16, 15, 14, 13, 12, 11, 10, 8.

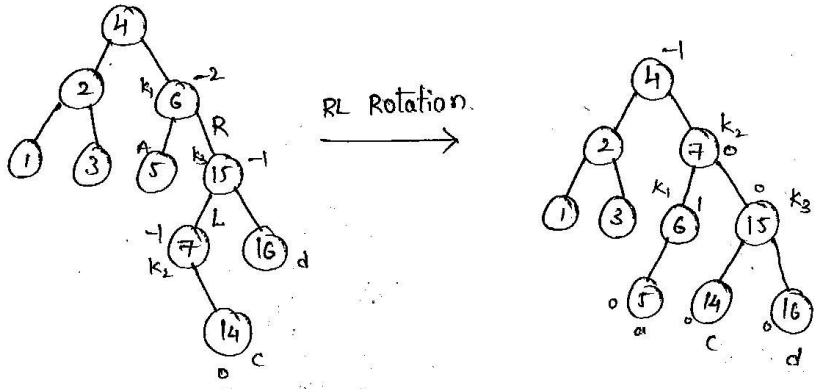
Insert 16 :-



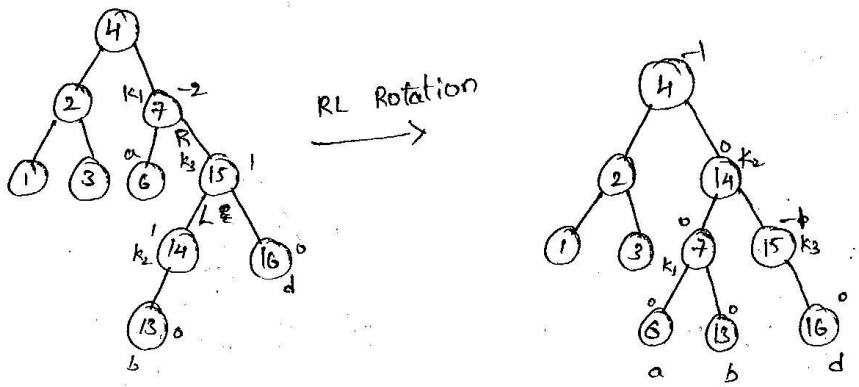
Insert 15:-



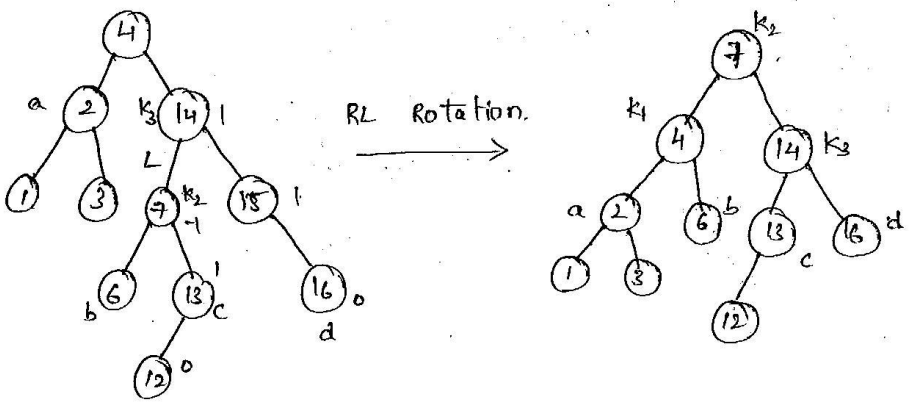
Insert 14:-



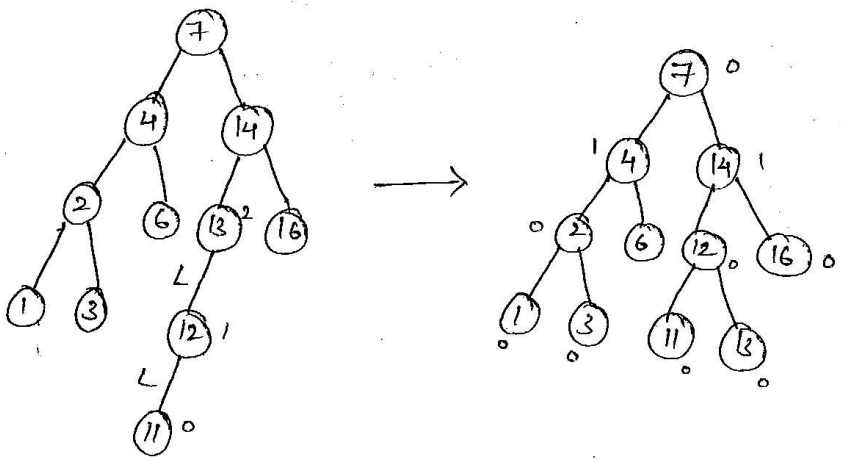
Insert -13 :-



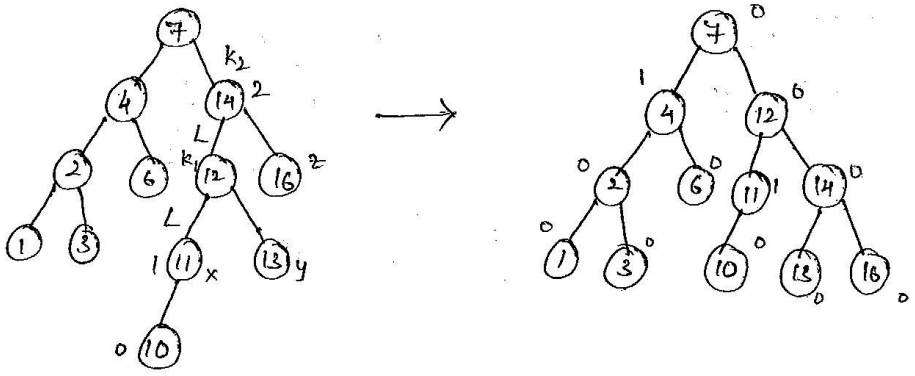
Insert 12 :-

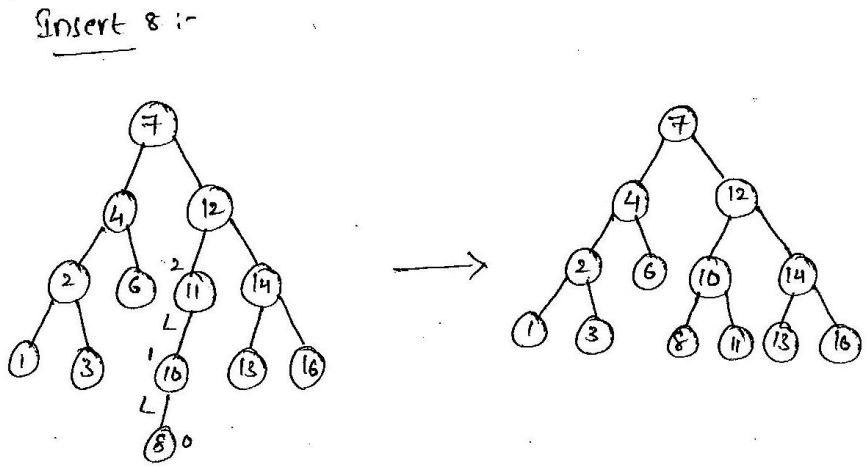


Insert 11 :-



Insert 10 :-





AVL TREE ALGORITHMS

```
Algorithm AVLInsert(root, newdata)
if(tree empty)
  insert newdata at root
else if(newdata<root)
  AVLInsert(left subtree,newdata)
  if(left subtree taller)
    leftBalance(root)
  end if
else
  AVLInsert(right subtree,newdata)
  if(right subtree taller)
    rightBalance(root)
  end if
end if
return root
end AVLInsert

Algorithm leftBalance(root)
if(left subtree high)  //Single rotation
  rotateRight(root)
else                   //Double rotation
  rotateLeft(left subtree)
  rotateRight(root)
end if
end leftBalance
```

```
Algorithm rightBalance(root)
if(right subtree high)  //Single rotation
  rotateLeft(root)
else                    //Double rotation
  rotateRight(right subtree)
  rotateLeft(root)
end if
end rightBalance

Algorithm rotateRight(root)
  exchange left subtree with right subtree of left subtree
  make left subtree new root
end rotateRight

Algorithm rotateLeft(root)
  exchange right subtree with left subtree of right subtree
  make right subtree new root
end rotateLeft

Algorithm AVLDelete(root,dltkey)
//This algorithm deletes a node from AVL Tree and rebalances if necessary
if(subtree empty)
  return NULL
end if
if(dltkey<root)
  set left subtree to AVLDelete(left subtree,dltkey)
  if(tree shorter)
    set root to deleteRightBalance(root)
  end if
else if(dltkey>root)
  set right subtree to AVLDelete(right subtree,dltkey)
  if(tree shorter)
    set root to deleteLeftBalance(root)
  end if
else
  save root
  if(no right subtree)
    return left subtree
  else if(no left subtree)
```

```
    return right subtree
  else
    find largest node on left subtree
    save largest key
    copy data in largest to root
    set left subtree to AVLDelete(left subtree,largest key)
    if(tree shorter)
     set root to deleteRightBalance(root)
    end if
end if
end if
return root
end AVLDelete

Algorithm deleteRightBalance(root)
//The [sub]tree is shorter after a deletion on the left branch
if(tree not balanced)
  set rightofRight to right subtree
  if(rightofRight left high)  //Double rotation required
    set leftofRight to left subtree of rightofRight
    //Rotate right then left
    right subtree=rotateRight(rightofRight)
    root=rotateLeft(root)
  else  //Single rotation required
    set root to rotateLeft(root)
  end if
end if
return root
End deleteRightBalance
```

AVL TREE DELETION

- The deletion operation in AVL tree is similar to deletion in BST, except that after deletion if AVL tree is unbalanced make that as balanced tree either by single or double rotation.
- Three possibilities for deletion:

(1) Assume the element to be deleted is 'x'

(a) If 'x' is a leaf node then remove 'x'. If the tree is unbalanced make it balanced.

(b) If 'x' has only one child then make parent of 'x' parent for its child and remove x.

(c) If 'x' has two children then consider any one of the following possibilities:

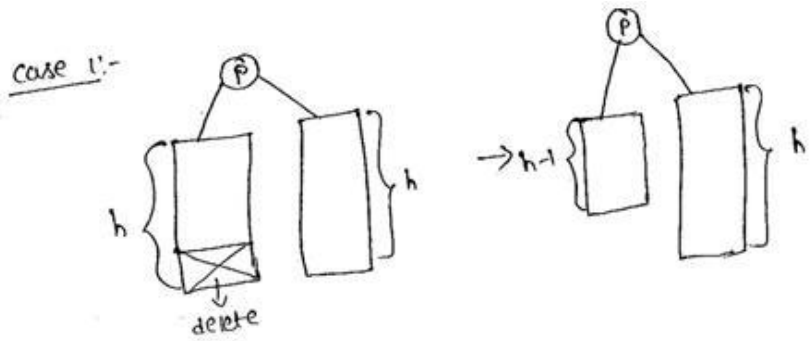
(i) Take the smallest value from the right sub tree. Let it occupy the position of 'x' and remove it from right sub tree.

(ii) Take the largest value from the left sub tree. Let it occupy the position of 'x' and remove it from left sub tree.

- After deleting the element check the balance factor from the node where deletion was applied to root and if any unbalances are there fix them by rotations.

Rotation Free Deletion

- Some deletions do not require any rotation. Because after deletion also the tree preserves AVL property.



- Previously the two sub trees have same height. But after deletion the height of sub trees will differ by '1'. So there will be no problem.

Ex:-

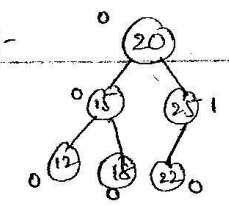
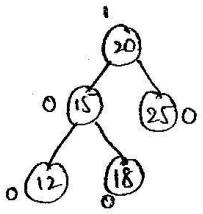


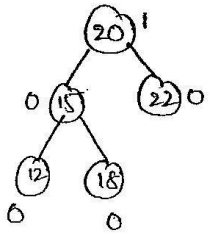
fig (a)

(a) Remove 22 from fig (a).



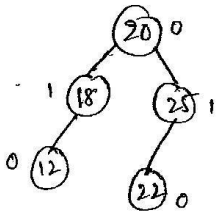
No rotation required.

(b) Remove 25 from fig(a)

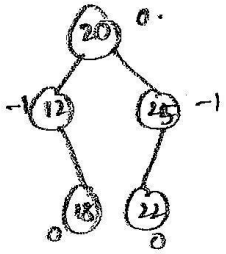


No rotation required.

(c) Remove 15 from fig(a).

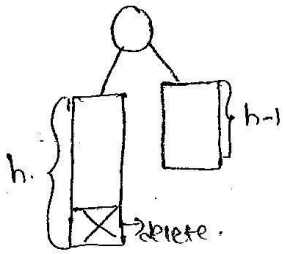


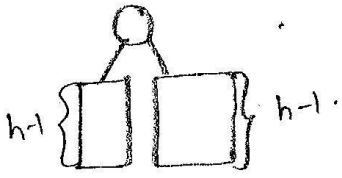
(or)



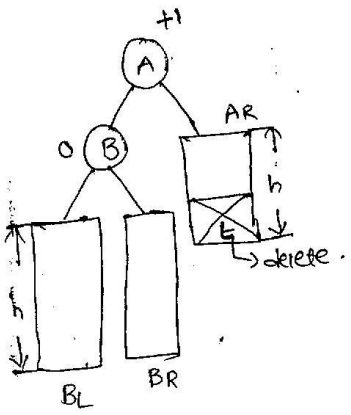
No rotations required

Case 2:- Here the balance factor of 'p' is not equal, but after deletion it will be equal.

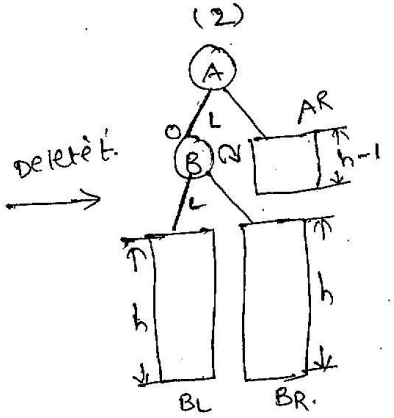




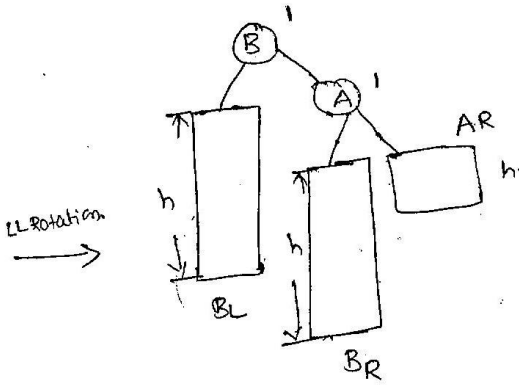
RO rotation :-



Balanced tree before deletion.

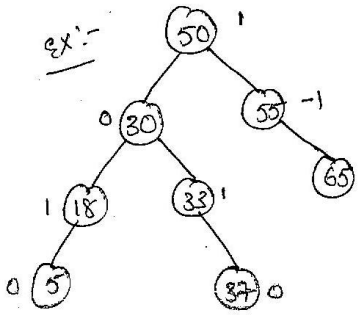


unbalanced tree After deleting it

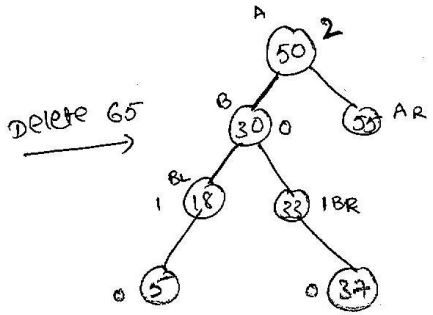


Balanced tree after deletion.

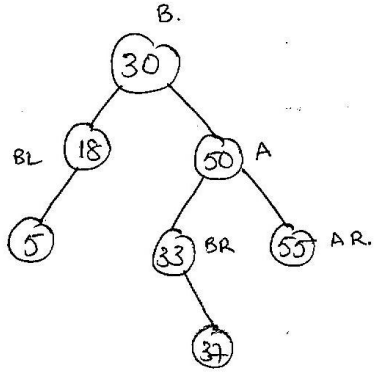
Ex:-



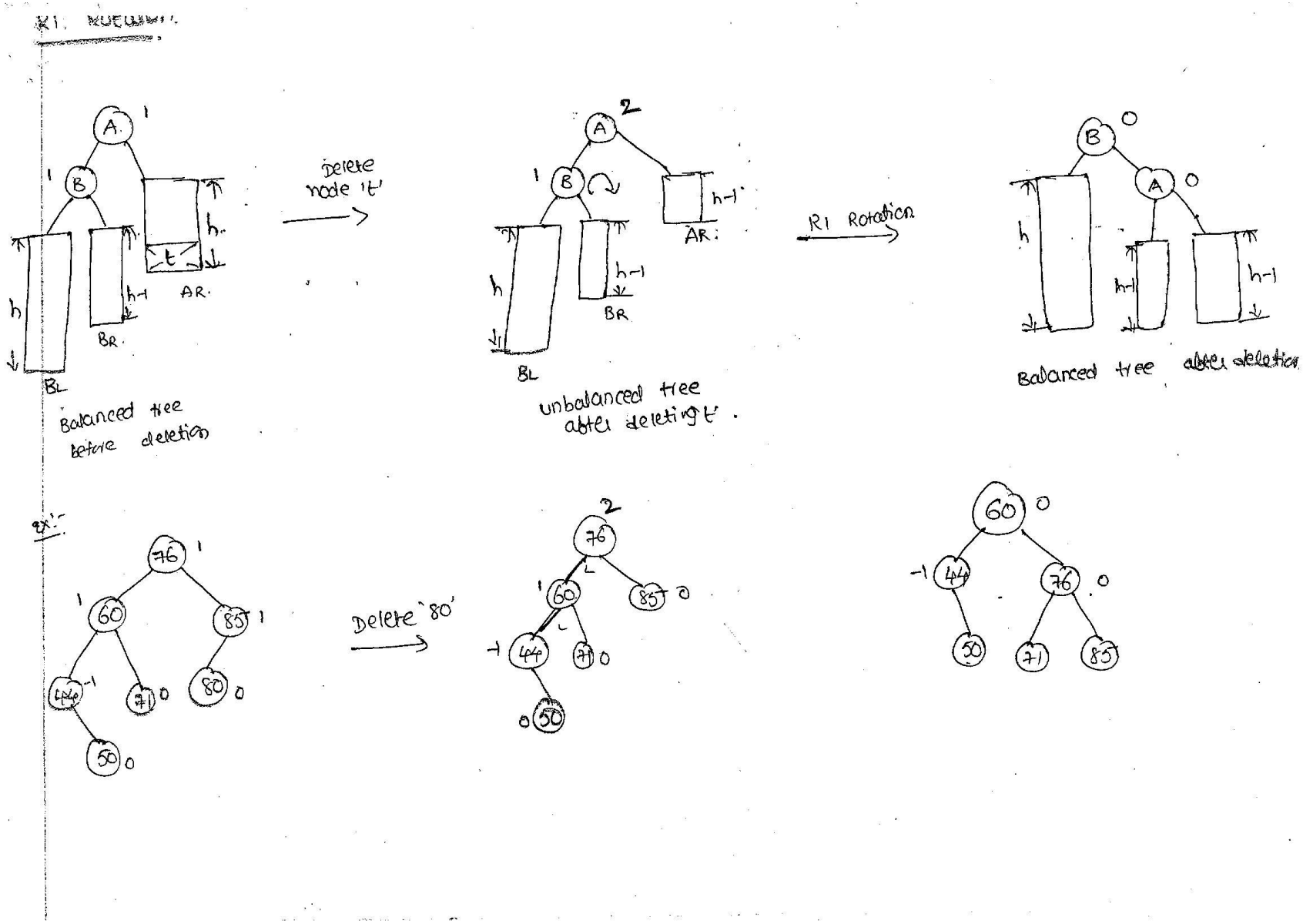
Balanced tree before deletion.



unbalanced tree after deletion.

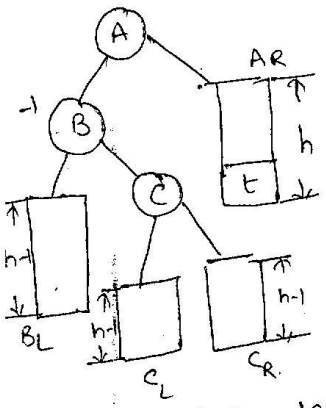


Balanced tree after deletion.



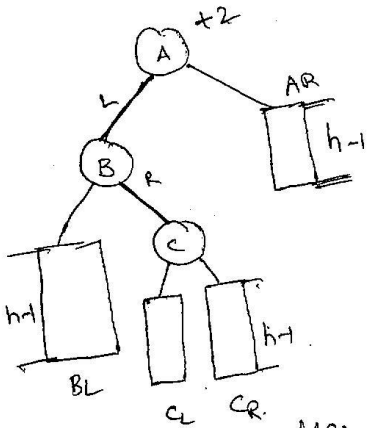
Visit https://www.tutorialsduniya.com for Notes, books, programs, question papers with solutions etc.

R-1 rotation:-



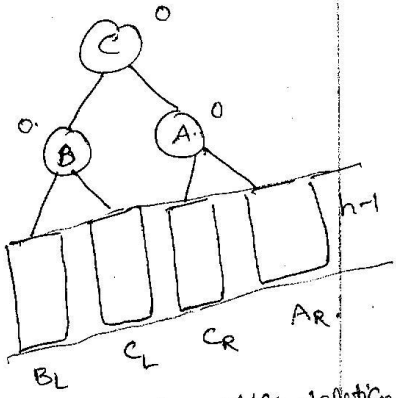
Balanced before deletion.

Delete 't'



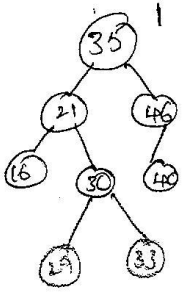
Unbalance after deleting t.

R-1 rotation

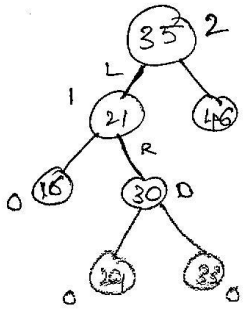


Balanced after deletion

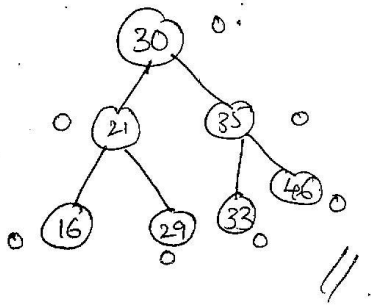
Ex:-

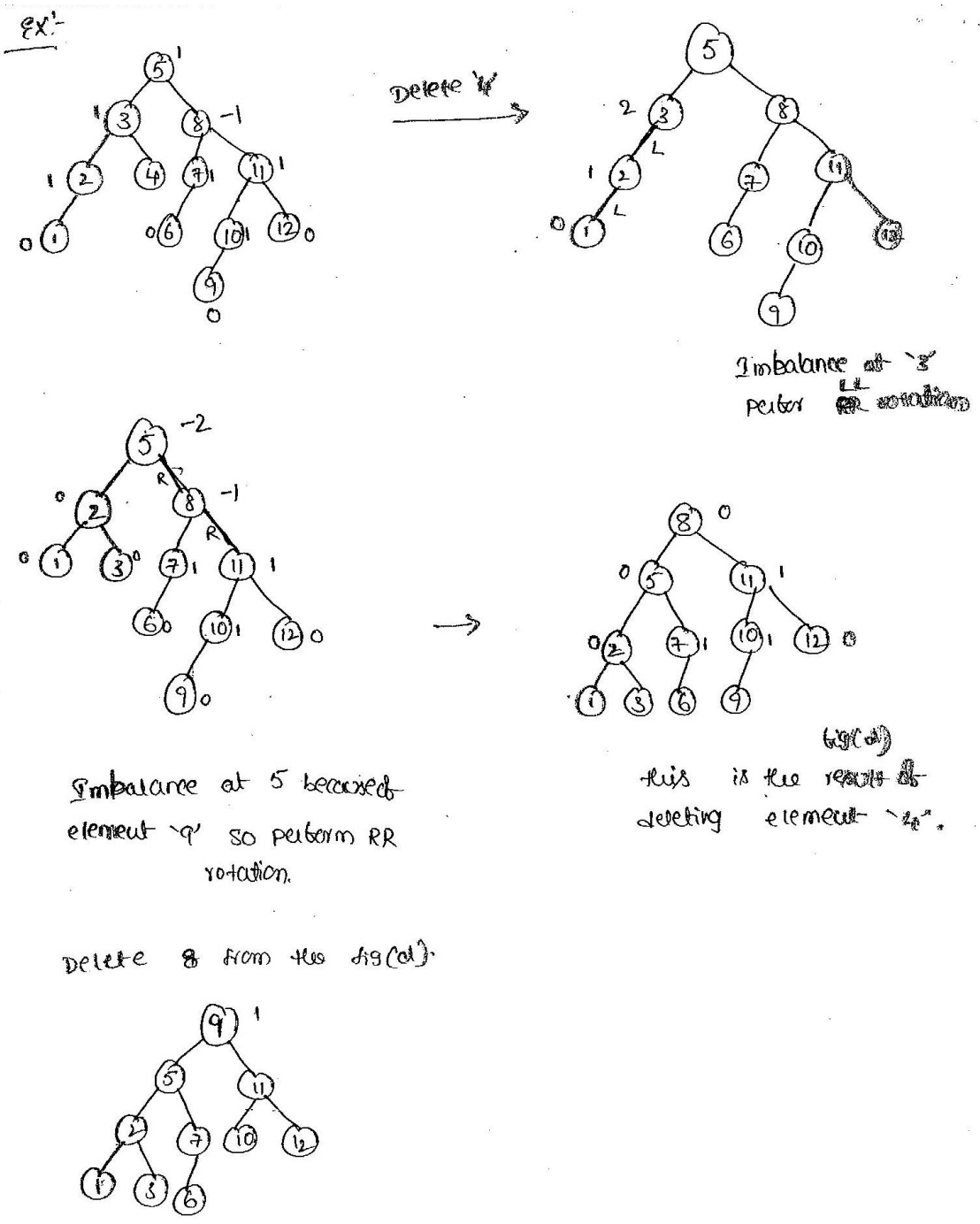


Delete '40'



Double Rotation (LR)





Maximum Height of an AVL Tree

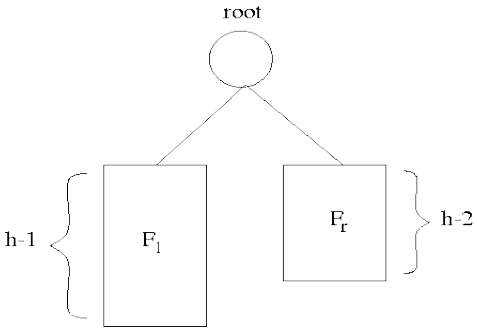
- The height of an AVL tree 'T' storing 'n' keys is O(logn).
- Assume n(h) is the minimum number of nodes in an AVL tree of height 'h'.
- An AVL tree of height 'h' contains:
  (a) A root node of height 'h'.
  (b) One AVL sub tree of height h-1 and the other AVL sub tree of height h-1 or h-2.

TutorialsDuniya.com

Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

Please Share these Notes with your Friends as well



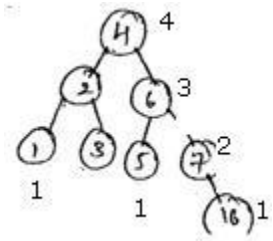
So n(h) can be defined as, n(h)=1+n(h-1)+n(h-2)

where, n(h) - no. of nodes in a tree, n(h-1) - no. of nodes in left sub tree, n(h-2) - no. of nodes in right sub tree, 1 - for root node.

n(h)=1+n(h-1)+n(h-2) for h≥3 (Since n(1)=1 and n(2)=2)

We know that n(h-1)≥n(h-2).

Ex: Suppose a tree has height h=4, then n(3)≥n(2)



- In this n(3) is 4 and n(2) is 3 i.e., left sub tree has 3 nodes, and right sub tree has 4 nodes.
- Since the height difference between left and right sub trees of an AVL tree is at most 1 in an AVL tree, the condition

By substituting n(h-1)≥n(h-2) in the equation and dropping 1 we get,

n(h)=n(h-1)+n(h-2)+1>2n(h-2)

n(h)>2n(h-2) (solving this recursively we get series of inequalities)

>2[2n((h-2)-2)]

>4n(h-4)

>8n(h-6)

>16n(h-8)

.

..

n(h)>2^i n(h-2i) (n(1)=1 and n(2)=2, we can consider 1 or 2 for h-2i)

Let i=$\frac{h}{2}-1$ (By taking h-2i=2)

n(h)>$2^{\frac{h}{2}-1}$.n(h-2($\frac{h}{2}-1$))

n(h)>$2^{\frac{h}{2}-1}$.n(h-$^{2h}_{2}$+1)

n(h)>$2^{\frac{h}{2}-1}$.n(h-h+2)

n(h)>$2^{\frac{h}{2}-1}$.n(2)

n(h)>$2^{h/2}$/2*n(2) [since n(2)=2 i.e., a tree with height 2 has minimum 2 nodes]

n(h)>$2^{h/2}$/2*2

n(h)>$2^{h/2}$

log (n(h))>h/2

2log (n(h))>h

h=O(log(n(h)) If number of nodes is n then h=O(logn).